

Axiomatizing Analog Algorithms

Olivier Bournez^{1*}, Nachum Dershowitz^{2**}, and Pierre Neron³

¹ Laboratoire d'Informatique de l'X (LIX), École Polytechnique, France

² School of Computer Science, Tel Aviv University, Ramat Aviv, Israel

³ French Network and Information Security Agency (ANSSI), France
 bournez@lix.polytechnique.fr nachum@cs.tau.ac.il pierre.neron@ssi.gouv.fr

Abstract. We propose a formalization of generic algorithms that includes analog algorithms. This is achieved by reformulating and extending the framework of abstract state machines to include continuous-time models of computation. We prove that every hybrid algorithm satisfying some reasonable postulates may be expressed precisely by a program in a simple and expressive language.

1 Introduction

In [14], Gurevich showed that any algorithm that satisfies three intuitive “Sequential Postulates” can be step-by-step emulated by an abstract state machine (ASM). These postulates formalize the following intuitions: (I) one is dealing with discrete deterministic state-transition systems; (II) the information in states suffices to determine future transitions and may be captured by logical structures that respect isomorphisms; and (III) transitions are governed by the values of a finite and input-independent set of ground terms. All notions of algorithms for “classical” *discrete-time* models of computation in computer science are covered by this formalization. This includes Turing machines, random-access memory (RAM) machines, and their sundry extensions. The geometric constructions in [18], for example, are loop-free examples of discrete-step continuous-space (real-number) algorithms. The ASM formalization also covers general discrete-time models evolving over continuous space like the Blum-Shub-Smale machine model [1].

However, capturing *continuous-time* models of computation is still a challenge, that is to say, capturing models of computation that operate in continuous (real) time and with real values. Examples of continuous-time models of computations include models of analog machines like the General Purpose Analog Computer (GPAC) of Claude Shannon [20], proposed as a mathematical

* This author’s research was partially supported by a French National Research Agency’s grant (ANR-15-CE40-0016-02).

** This author’s research benefited from a fellowship at the Institut d’Études Avancées de Paris (France), with the financial support of the French National Research Agency’s “Investissements d’avenir” program (ANR-11-LABX-0027-01 Labex RFIEA+).

model of the Differential Analyzers, built for the first time in 1931 [7], and used to solve various problems ranging from ballistics to aircraft design – before the era of the digital computer [16]. Others include Pascal’s 1642 *Pascaline*, Hermann’s 1814 *Planimeter*, as well as Bill Phillips’ 1949 water-run *Financephalograph*. Continuous-time computational models also include neural networks and systems built using electronic analog devices. Such systems begin in some initial state and evolve over time; results are read off from the evolving state and/or from a terminal state. More generally, determining which systems can actually be considered to be computational models is an intriguing question and relates to philosophical discussions about what constitutes a programmable machine. Continuous-time computation theory is far less understood than its discrete-time counterpart [4]. Another line of development of continuous-time models was motivated by hybrid systems, particularly by questions related to the hardness of their verification and control. In hybrid systems, the dynamics change in response to changing conditions, so there are discrete transitions as well as continuous ones. Here, models are not seen as necessarily modeling analog machines, but, rather, as abstractions of systems about which one would like to establish properties or derive verification algorithms [4]. Some work on ASM models dealing with continuous-time systems has been accomplished for specific cases [8,9]. Rust [19] specifies forms of continuous-time evolution based on ASMs using infinitesimals. However, we find that a comprehensive framework capturing general analog systems is still wanting.

Our goal is to capture all such analog and hybrid models within one uniform notion of computation and of algorithm. To this end, we formalize a generic notion of continuous-time algorithm. The proposed framework is an extension of [14], as discrete-time algorithms are a simple special case of analog algorithms. (The initial attempt [5] was not fully satisfactory, as no completeness theorem nor general-form result was obtained. Here, we indeed achieve both.) We provide postulates defining continuous-time algorithms, in the spirit of those of [14], and we prove some completeness results. We define a simple notion of an analog ASM program and prove that all models satisfying the postulates have corresponding analog programs (Lemma 16 and Theorem 26). Furthermore, we provide conditions guaranteeing that said program is unique up to equivalence (Theorem 27 and Corollary 28). All of this seamlessly extends the results of [14] to analog and hybrid systems. The proposed framework covers all classes of continuous-time systems that can be modeled by ordinary differential equations or have hybrid dynamics, including the models in [4] and the examples in [5]. It is a first step towards a general understanding of computability theory for continuous-time models, taken in the hope that it will also lead to a formalization of a “Church-Turing thesis” for analog systems in the spirit of what has been achieved for discrete-time models [2,10,3]. Systems with continuous input signals and other means of specifying continuous behavior are left for future work.

Some of our ideas were inspired by the way the semantics of hybrid systems are given in the approach of Platzer [17]. Among attempts at studying the semantics of analog systems within a general framework is [22]. Recent results on

comparing analog models include [11]. Soundness and (relative) completeness results for a programming language with infinitesimals have also been obtained in [21]. Applications to verification have been explored [15].

2 General Algorithms

We want to generalize the notion of algorithms introduced by Gurevich in [14] in order to capture not only the sequential case but also continuous behavior. (For lack of place, we assume some familiarity with [14].) However, when evolving continuously, an algorithm can no longer be viewed as a discrete sequence of states, and we need a notion of evolution that can capture both kinds of behavior. This is based on a notion of a *timeline* that corresponds to algorithm execution.

Definition 1 (Time). Time \mathbb{T} corresponds to a totally ordered monoid: there is an associative binary operation $+$, with some neutral element 0 , and a total relation \leq preserved by $+$: $t \leq t'$ implies $t + t'' \leq t' + t''$ for all $t'' \in \mathbb{T}$.

An element of \mathbb{T} will be called a *moment*. Examples of time \mathbb{T} are $\mathbb{R}^{\geq 0}$ and \mathbb{N} . As expected, $t < t'$ will mean $t \leq t'$ but not $t = t'$.

Definition 2 (Timeline). A timeline is a subset of \mathbb{T} containing 0 . We let \mathbb{I} denote the set of all timelines.

For a moment $i \in I$ of timeline I , we write $\text{Jump}(i)$ if there exists $t \in I$ with $i < t$, and there is no $t' \in I$ with $i < t' < t$. We write $\text{Flow}(i)$ otherwise: that means that for all t , $i < t$, there is some in-between $t' \in I$ with $i < t' < t$. A moment i with $\text{Jump}(i)$ is meant to indicate a discrete transition. In this case, we write i^+ for the smallest t greater than i . A timeline I is non-Zeno if for any moment $i \in I$, there is a finite number of moments $j \leq i$ with $\text{Jump}(j)$. \mathbb{I} is non-Zeno if all its timelines are.

For timelines $\mathbb{I} = \mathbb{R}^{\geq 0}$, for instance, we have $\text{Flow}(i)$ for all $i \in \mathbb{I}$. For $\mathbb{I} = \mathbb{N}$, we have $\text{Jump}(i)$ for all $i \in \mathbb{I}$, and $i^+ = i + 1$. We intend (for hybrid systems, in particular) to also consider timelines mixing both properties, that is, with $\text{Flow}(i)$ for some i and $\text{Jump}(i)$ for other i . Formally building such timelines is easy (for example $\bigcup_{n \in \mathbb{N}} [n, n + 0.5]$). All these examples are non-Zeno.

Definition 3 (Truncation). Given a timeline $I \in \mathbb{I}$ and a moment i of I , the truncated timeline $I[i]$ is the timeline defined by $I[i] = \{t \mid i + t \in I\}$.

With timelines in hand, we can define hybrid dynamical systems.

Definition 4 (Dynamical System). A dynamical system $\langle \mathcal{S}, \mathcal{S}_0, \iota, \varphi \rangle$ consists of the following: (a) a nonempty set (or class) \mathcal{S} of states; (b) a nonempty subset (or subclass) $\mathcal{S}_0 \subseteq \mathcal{S}$, called initial states; (c) a timeline map $\iota : \mathcal{S} \rightarrow \mathbb{I}$, with \mathbb{I} non-Zeno; (d) a trajectory map $\varphi : (X : \mathcal{S}) \times \iota(X) \rightarrow \mathcal{S}$. We require that, for any state X and moments $i, i + i' \in \iota(X)$, one has

$$\iota(X, 0) = X, \quad \iota(\varphi(X, i)) = \iota(X)[i], \quad \varphi(X, i + i') = \varphi(\varphi(X, i), i').$$

Together, the *timeline* and *trajectory* maps associate to each state its future evolution. For a state X , $\iota(X)$ defines the timeline corresponding to the system behavior starting from X , and $\varphi(X)$ defines its concrete evolution by associating to each moment in $\iota(X)$ its corresponding state. The third condition ensures that evolution during $i + i'$ is similar to first evolving during i and then during i' ; the preceding condition ensures a similar property for timelines (and ensures consistency of the last condition).

Postulate I. *An algorithm is a dynamical system.*

A vocabulary \mathcal{V} is a finite collection of fixed-arity (possibly nullary) function symbols, some functions of which may be tagged *relational*. A term whose outermost function symbol is relational is termed *Boolean*. We assume that \mathcal{V} contains the scalar (nullary) function *true*. A (first-order) *structure* X of vocabulary \mathcal{V} is a nonempty set S , the *base set* (*domain*) of X , together with interpretations of the function symbols in \mathcal{V} over S : A j -ary function symbol f is interpreted as a function, denoted $\llbracket f \rrbracket_X$, from S^j to S . Elements of S are also called elements of X , or *values*. Similarly, the interpretation of a term $f(t_1, \dots, t_n)$ in X is recursively defined by $\llbracket f(t_1, \dots, t_n) \rrbracket_X = \llbracket f \rrbracket_X (\llbracket t_1 \rrbracket_X, \dots, \llbracket t_n \rrbracket_X)$.

Let X and Y be structures of the same vocabulary \mathcal{V} . An *isomorphism* from X onto Y is a one-to-one function ζ from the base set of X onto the base set of Y such that $f(\zeta x_1, \dots, \zeta x_j) = \zeta x_0$ in Y whenever $f(x_1, \dots, x_j) = x_0$ in X .

Definition 5 (Abstract Transition System). *An abstract transition system is a dynamical system whose states \mathcal{S} are (first-order) structures over some finite vocabulary \mathcal{V} , such that the following hold:*

- (a) *States are closed under isomorphism, so if $X \in \mathcal{S}$ is a state of the system, then any structure Y isomorphic to X is also a state in \mathcal{S} , and Y is an initial state if X is.*
- (b) *Transformations preserve the base set: that is, for every state $X \in \mathcal{S}$, for any $i \in \iota(X)$, $\varphi(X, i)$ has the same base set as X .*
- (c) *Transformations respect isomorphisms: if $X \cong_\zeta Y$ is an isomorphism of states $X, Y \in \mathcal{S}$, then $\iota(X) = \iota(Y)$ and for all $i \in \iota(X)$, $X_i \cong_\zeta Y_i$, where $X_i = \varphi(X, i)$, and $Y_i = \varphi(Y, i)$.*

Postulate II. *An algorithm is an abstract transition system.*

When $\iota(X)$ is \mathbb{N} (or order-isomorphic to \mathbb{N}) for all X , this corresponds precisely to the concepts introduced by [14], considering that $\varphi(X, n) = \tau^{[n]}(X)$.

It is convenient to think of a structure X as a memory of some kind: If f is a j -ary function symbol in vocabulary \mathcal{V} , and \bar{a} is a j -tuple of elements of the base set of X , then the pair (f, \bar{a}) is called a *location*. We denote by $\llbracket f(\bar{a}) \rrbracket_X$ its interpretation in X , i.e. $\llbracket f \rrbracket_X (\bar{a})$. If (f, \bar{a}) is a location of X and b is an element of X then (f, \bar{a}, b) is an *update* of X . When Y and X are structures over the same domain and vocabulary, $Y \setminus X$ denotes the set of updates $\Delta^+ = \{(f, \bar{a}, \llbracket f(\bar{a}) \rrbracket_Y) \mid \llbracket f(\bar{a}) \rrbracket_Y \neq \llbracket f(\bar{a}) \rrbracket_X\}$.

We want instantaneous evolution to be describable by updates:

Definition 6. An infinitesimal generator is (a) a function Δ that maps states X to a set $\Delta(X)$ of updates, and (b) preserves isomorphisms: if $X \cong_{\zeta} Y$ is an isomorphism of states $X, Y \in \mathcal{S}$, then for all updates $(f, \bar{a}, b) \in \Delta(X)$, we have an isomorphic update $(f, \zeta \bar{a}, \zeta b) \in \Delta(Y)$.

We write $\text{Jump}(X)$ and say that X is a *jump* when $\text{Jump}(0)$ in timeline $\iota(X)$; otherwise, we write $\text{Flow}(X)$ and say that it is a *flow*. For states X with $\text{Jump}(X)$, the following is natural:

Definition 7. The update generator is the infinitesimal generator defined on jump states X as $\Delta(X) = \Delta^+(X)$, where $\Delta^+(X)$ stands for $\varphi(X, 0^+) \setminus X$.

To deal with flow states, we will also define some corresponding infinitesimal generator Δ_{ψ} . Before doing so, let's see how to go from semantics to generators.

An *initial evolution* over S is a function whose domain of definition is a timeline and whose range is S . An initial evolution is said to be *initially constant* if it has a constant prefix: that is to say, there is some $0 < t$ such that the function is constant over $[0 \dots t]$.

Definition 8 (Semantics). A semantics ψ over a class \mathcal{C} of sets S is a partial function mapping initial evolutions over some $S \in \mathcal{C}$ to an element of S .

Remark 9. When $\mathbb{T} = \mathbb{R}^{\geq 0}$, an example of semantics over the class of sets S containing \mathbb{R} is the derivative ψ_{der} , mapping a function f to its derivative at 0 when that exists. When $\mathbb{T} = \mathbb{N}$, an example of semantics over the class of all sets would be the function $\psi_{\mathbb{N}}$ mapping f to $f(1)$. More generally, when $0 \in \mathbb{T}$ is such that $\text{Jump}(0)$, an example of semantics over the class of all sets is the function $\psi_{\mathbb{N}}$ mapping f to $f(0^+)$.

Consider a semantics ψ over a class of sets S . Let X be a state whose domain is in the class and a location (f, \bar{a}) of X . Denote by $\text{Evolution}(X, (f, \bar{a}))$ the corresponding initial evolution: that is to say, the function given formally by $\text{Evolution}(X, (f, \bar{a})) : t \mapsto \llbracket f(\bar{a}) \rrbracket_{\varphi(X, t)}$ for $0 \leq t \leq I_1, t \in \iota(X)$, for some $I_1 \in \iota(X)$, with $I_1 = 0^+$ for a jump. We use $\psi[X, f, \bar{a}]$ to denote the image of this evolution under ψ (when it exists).

Definition 10 (Infinitesimal generator associated with ψ). The infinitesimal generator associated with ψ , maps each state X , such that $\psi[X, f, \bar{a}]$ is defined for all locations, to the set: $\Delta_{\psi}(X) = \{(f, \bar{a}, \psi[X, f, \bar{a}]) \mid (f, \bar{a}) \text{ is a location of } X, \text{Evolution}(X, (f, \bar{a})) \text{ is not initially constant}\}$.

The update generator Δ^+ (see Definition 7) is the infinitesimal generator associated with the semantics $\psi_{\mathbb{N}}$ (of Remark 9) over flow states.

From now on, we assume that some semantics ψ is fixed to deal with flow states. It could be ψ_{der} , but it could also be another one (for example: talking about integrals or built using infinitesimals as in [19]). We denote by Δ_{ψ} the associated infinitesimal generator.

We are actually discussing algorithms relative to some ψ , and to be more precise, we should be referring to ψ -algorithms. The point is that not every infinitesimal generator is appropriate and that appropriateness is actually relative to a time domain and to the class of allowed dynamics over this time domain. To see this, keep in mind that – when Δ_ψ corresponds to derivative – to be able to talk about derivatives, one implicitly restricts oneself to dynamics that are differentiable, hence non-arbitrary. In other words, one is restricting to a particular class of possible dynamics, and not all dynamics are allowed. Restricting to other classes of dynamics (for example, analytic ones) may lead to different notions of algorithm.

From the update generator Δ^+ and Δ_ψ , we build a generator also tagging states by the fact that they correspond to a jump or a flow:

Definition 11 (Generator of a State). *We define the tagged generator of a state X , denoted $\Delta_t(X)$, as a function that maps state X to $\{\mathcal{F}\} \times \Delta_\psi(X)$ when $\text{Flow}(X)$ and $\Delta_\psi(X)$ is defined and to $\{\mathcal{J}\} \times \Delta^+(X)$ when $\text{Jump}(X)$.*

Let T be a set of ground terms. We say that states X and Y *coincide* over T , if $\llbracket s \rrbracket_X = \llbracket s \rrbracket_Y$ for all $s \in T$. This will be abbreviated $X =_T Y$. The fact that X and Y coincide over T implies that X and Y necessarily share some common elements in their respective base sets, at least all the $\llbracket s \rrbracket_X$ for $s \in T$.

An algorithm should have a finite imperative description. Intuitively, the evolution of an algorithm from a given state is only determined by inspecting part of this state by means of the terms appearing in the algorithm description. The following corresponds to the *Bounded Exploration* postulate in [14].

Postulate III. *For any algorithm, there exists a finite set T of ground terms over vocabulary \mathcal{V} such that for all states X and Y that coincide for T , $\Delta_t(X)$ and $\Delta_t(Y)$ both exist and $\Delta_t(X) = \Delta_t(Y)$.*

A ground term of T is a *critical term* and a *critical element* is the value (interpretation) of a critical term.

Definition 12 (Analog Algorithm). *An algorithm is an object satisfying Postulates I through III.*

3 Characterization Theorem

We now go on to define the rules of our programs (adding to those of ASM programs in [14]).

Definition 13. – **Update Rule:** *An update rule of vocabulary \mathcal{V} has the form $f(t_1, \dots, t_j) := t_0$ where f is a j -ary function symbol in \mathcal{V} and t_1, \dots, t_j are ground terms over \mathcal{V} .*

– **Parallel Update Rule:** *If R_1, \dots, R_k are update rules of vocabulary \mathcal{V} , then*

```

par
  R1
  R2
  ...
  Rk
endpar

```

is a parallel update rule of vocabulary \mathcal{V} .

$\Delta_t(R_i, X)$ denotes the interpretation of a rule R in state X and is defined as expected: If R is an update rule $f(t_1, \dots, t_j) := t_0$ then $\Delta_t(R, X) = \{\mathcal{J}\} \times (f, (\llbracket t_i \rrbracket_X, \dots, \llbracket t_j \rrbracket_X), \llbracket t_0 \rrbracket_X)$ and when R is **par** R_1, \dots, R_k **endpar** then $\Delta_t(R, X) = \{\mathcal{J}\} \times (d_1 \cup \dots \cup d_k)$ where $\Delta_t(R_i, X) = \{\mathcal{J}\} \times d_i$ for all i .

Next, we introduce rules to deal with *Flows*.

Definition 14. – **Basic Continuous Rule:** A basic continuous rule of vocabulary \mathcal{V} has the form $\text{DYNAMIC}(f(t_1, \dots, t_j), t_0)$ where f is a symbol of arity j and t_0, t_1, \dots, t_j are ground terms of vocabulary \mathcal{V} .
– **Flow Rule:** If R_1, \dots, R_k are basic continuous rules of vocabulary \mathcal{V} , then

```

flow
  R1
  R2
  ...
  Rk
endflow

```

is a flow rule of vocabulary \mathcal{V} .

Their semantics are then defined as follows. If R is a basic continuous rule $\text{DYNAMIC}(f(t_1, \dots, t_j), t_0)$ then $\Delta_t(R, X) = \{\mathcal{F}\} \times \{(f, (a_1, \dots, a_j), a_0)\}$ where each $a_i = \llbracket t_i \rrbracket_X$. If R is a flow rule with constituents R_1, \dots, R_k , then $\Delta_t(R, X) = \{\mathcal{F}\} \times (d_1 \cup \dots \cup d_k)$ where $\Delta_t(R_i, X) = \{\mathcal{F}\} \times d_i$.

Finally, we allow conditionals:

Definition 15. – **Selection Rule:** If φ is a ground boolean term over vocabulary \mathcal{V} and R_1 and R_2 are rules of vocabulary \mathcal{V} then:

```

if  $\varphi$  then
  R1
else
  R2
endif

```

is a rule of vocabulary \mathcal{V} .

Given such a rule R and a state X , if φ evaluates to *true* (the interpretation of scalar function *true*) in X then $\Delta_t(R, X) = \Delta_t(R_1, X)$ else $\Delta_t(R, X) = \Delta_t(R_2, X)$.

An ASM program of vocabulary \mathcal{V} is a rule of vocabulary \mathcal{V} . The first key result is the following, which can be seen as a completeness result.

Theorem 16 (Completeness). *For every algorithm of vocabulary \mathcal{V} , there is an ASM program Π over \mathcal{V} with the identical behavior: $\Delta_t(\Pi, X) = \Delta_t(X)$ for all states X .*

4 Proof of Theorem 16

Before turning to the proof of our main theorem, we reformulate and extend several of the constructions in [14].

Lemma 17 ([14, Lemma 6.2]). *Consider an algorithm A , consider a state X of A and assume $\text{Jump}(X)$. By definition, $\Delta_t(X) = \{\mathcal{J}\} \times \Delta^+(X)$.*

Consider $(f, a_1, \dots, a_j, a_0)$, an update of $\Delta^+(X)$. Then all elements a_0, a_1, \dots, a_j are critical elements of X , that is, they correspond to values (interpretations) of critical terms.

Proof. The proof proceeds by contradiction. Assume that some a_k does not correspond to the value of any critical term. One can easily consider a structure Y isomorphic to X which is obtained from X by replacing a_k with a fresh element b . By Postulate II, Y is a state and $\llbracket t \rrbracket_Y = \llbracket t \rrbracket_X$ for every critical term t . By Postulate III, we know that $\text{Jump}(Y)$, and we must have: $\Delta_t(Y) = \{\mathcal{J}\} \times \Delta^+(Y) = \{\mathcal{J}\} \times \Delta^+(X)$. By Postulate II, a_k does not occur in (the base set of) $\varphi(Y, 0^+)$ either. Hence, it cannot occur in $\Delta^+(Y) = \varphi(Y, 0^+) \setminus Y$. This gives the desired contradiction. \square

Lemma 18 (Generalization of [14, Lemma 6.2]). *Consider an algorithm A and assume $\text{Flow}(X)$. Then by definition $\Delta_t(X) = \{\mathcal{F}\} \times \Delta_\psi(X)$.*

Consider $(f, a_1, \dots, a_j, a_0)$, an element of $\Delta_\psi(X)$. Then all elements a_0, a_1, \dots, a_j are critical elements of X , that is, they correspond to values of critical terms.

Proof. The proof proceeds by contradiction. Assume that some a_k does not correspond to the value of any critical term. One can easily consider a structure Y isomorphic to X which is obtained from X by replacing a_k with a fresh element b .

By Postulate II, Y is a state. Observe that $\llbracket t \rrbracket_Y = \llbracket t \rrbracket_X$ for every critical term t .

By Postulate III, we know that $\text{Flow}(Y)$, and we must have:

$$\Delta_t(Y) = \{\mathcal{F}\} \times \Delta_\psi(Y) = \{\mathcal{F}\} \times \Delta_\psi(X).$$

By Postulate II, a_k does not occur in (the base set of) Y . Hence it cannot occur in $\Delta_\psi(Y)$, since by Definition 6 elements in $\Delta_\psi(Y)$ are elements of the base set of Y . This gives the desired contradiction. \square

The following follows directly from Lemmas 17 and 18.

Corollary 19 (Corollary 6.6 of [14]). *For every state X , there exists a rule R^X such that $\Delta_t(X) = \Delta_t(R^X, X)$.*

We now generalize some of the other lemmas from [14] to apply to our more general setting.

Lemma 20 (Generalization of [14, Lemma 6.7]). *If states X and Y coincide over the set T of critical terms, then:*

$$\Delta_t(R^X, Y) = \Delta_t(Y).$$

Proof. We have $\Delta_t(R^X, Y) = \Delta_t(R^X, X) = \Delta_t(X) = \Delta_t(Y)$. The first equality holds because R^X involves only critical terms and because critical terms have the same values in X and Y . The second equality holds by the definition of R^X (that is to say, this is the previous corollary). The third equality holds because of the choice of T and because X and Y coincide over T . \square

Lemma 21 (Generalization of [14, Lemma 6.8]). *Suppose that X, Y are states and that $\Delta_t(R^X, Z) = \Delta_t(Z)$ for some state Z isomorphic to Y then:*

$$\Delta_t(R^X, Y) = \Delta_t(Y).$$

Proof. Let ζ be an isomorphism from Y onto an appropriate Z . Extend ζ to tuples, locations, updates and set of updates. It is easy to check that $\zeta(\Delta_t(R^X, Y)) = \Delta_t(R^X, Z)$. By the choice of Z , $\Delta_t(R^X, Z) = \Delta_t(A, Z)$.

By Definition 6, generators preserve isomorphisms, thus $\Delta_t(A, Z) = \zeta(\Delta_t(A, Y))$ and then $\zeta(\Delta_t(R^X, Y)) = \zeta(\Delta_t(A, Y))$. It remains to apply ζ^{-1} to both sides of the last equality. \square

At each state X , the equality relation between critical elements induces an equivalence relation

$$E_X(t_1, t_2) \text{ iff } \llbracket t_1 \rrbracket_X = \llbracket t_2 \rrbracket_X$$

over critical terms.

States X and Y are T -similar if $E_X = E_Y$.

Lemma 22 (Generalization of [14, Lemma 6.9]). *Let X be a state. Then, for every state Y that is T -similar to X , we have:*

$$\Delta_t(R^X, Y) = \Delta_t(Y).$$

Proof. Replace every element of Y that belongs to X with a fresh element. This gives a structure Z_1 that is isomorphic to Y and disjoint from X . By Postulate II, Z_1 is a state. Since Z_1 is isomorphic to Y , it is T -similar to Y and therefore T -similar to X .

Let Z_2 be the structure isomorphic to Z_1 that is obtained from Z_1 by replacing $\llbracket t \rrbracket_Y$ with $\llbracket t \rrbracket_X$ for all critical term t (the definition of Z_2 is coherent because X and Z_1 are T -similar). By Lemma 20, we have $\Delta_t(R^X, Z_2) = \Delta_t(Z_2)$.

Since Z_2 is isomorphic to Z_1 isomorphic to Y , then Z_2 is isomorphic to Y and by Lemma 21, we conclude $\Delta_t(R^X, Y) = \Delta_t(Y)$. \square

By previous Lemma 22, for every state X , there exists a boolean term φ^X that evaluates to **true** in a structure Y if and only if Y is T -similar to X . Indeed, the desired term asserts that the equality relation on the critical terms is exactly the equivalence relation E_X .

Since there are only finitely many critical terms, there are only finitely many possible equivalence relations E_X . Hence there exists a finite set $\{X_1, \dots, X_m, Y_1, \dots, Y_n\}$ of states such that every state is T -similar to one of the state X_i or Y_i , and such that $Jump(X_i)$ and $Flow(Y_i)$ for all i (recall that the property of being $Flow()$ is preserved by T -similarity from the previous lemma). States $\{X_1, \dots, X_m, Y_1, \dots, Y_n\}$ can be chosen mutually exclusive, that is to say in different equivalence relations. Boolean terms $(\varphi^{X_i})_i$ and $(\varphi^{Y_i})_i$ then realize a partition of the set of states.

We can then go to the proof of Theorem 16: Let $X_1, \dots, X_m, Y_1, \dots, Y_n$ be as above. The desired Π is

```

if  $\varphi^{X_1}$  then
   $R^{X_1}$ 
else
  if  $\varphi^{X_2}$  then
     $R^{X_2}$ 
  else
    ...
    if  $\varphi^{X_m}$  then
       $R^{X_m}$ 
    else
      if  $\varphi^{Y_1}$  then
         $R^{Y_1}$ 
      else
        if  $\varphi^{Y_2}$  then
           $R^{Y_2}$ 
        else
          ...
          if  $\varphi^{Y_{n-1}}$  then
             $R^{Y_{n-1}}$ 
          else
             $R^{Y_n}$ 
          endif
        endif
      endif
    endif
  endif
endif
endif
endif
endif

```

where the R^{X_i} are (possibly parallel) update rules, and the R^{Y_i} are flow rules.

5 Extended Statements

We are now very close to formulating our other theorems. First we define an abstract state machine relative to semantics ψ .

Definition 23. A ψ -abstract state machine B comprises the following: (a) an ASM program Π ; (b) a set \mathcal{S} of (first-order) structures over some finite vocabulary \mathcal{V} closed under isomorphisms, and a subset $\mathcal{S}_0 \subseteq \mathcal{S}$ closed under isomorphisms; (c) a map ι and a map φ such that $\langle \mathcal{S}, \mathcal{S}_0, \iota, \varphi \rangle$ is an algorithm, where Δ_ψ is fixed to be the infinitesimal generator associated with ψ , and for all states X in \mathcal{S} , $\Delta_t(\Pi, X) = \Delta_t(X)$.

By definition, a ψ -abstract state machine B satisfies all the postulates and hence is an algorithm.

Definition 24. An ASM program Π is ψ -solvable for a set \mathcal{S} of (first-order) structures over some finite vocabulary \mathcal{V} closed under isomorphisms and a subset $\mathcal{S}_0 \subseteq \mathcal{S}$ closed under isomorphisms if there exists a unique ι and φ such that $(\Pi, \mathcal{S}, \mathcal{S}_0, \iota, \varphi)$ is a ψ -abstract state machine.

Definition 25. A semantics ψ is unambiguous if for all sets \mathcal{S} of (first-order) structures over some finite vocabulary \mathcal{V} closed under isomorphisms, and for all subsets $\mathcal{S}_0 \subseteq \mathcal{S}$ closed under isomorphisms, whenever there exists some ι and φ such that $(\Pi, \mathcal{S}, \mathcal{S}_0, \iota, \varphi)$ is a ψ -abstract state machine, then ι and φ are unique.

Our main results follow.

Theorem 26. For every ψ -definable algorithm A , there exists an equivalent ψ -abstract state machine B .

Proof. By construction, A is a hybrid dynamical system such that $\Delta_t(A, X) = \Delta_t(\Pi, X)$ for some Π given by previous discussions. Set the states of B to be the states of A and the initial states of B to be the initial states of A . \square

Theorem 27. Assume that ψ is unambiguous. For every ψ -definable algorithm A , there exists a unique equivalent ψ -abstract state machine B with same states and initial states.

Proof (of Theorem 27). This is exactly the same proof as for Theorem 26. Unicity comes from the definition of unambiguity. \square

Corollary 28. Assume that ψ is unambiguous. For every ψ -definable algorithm A , there exists an equivalent ψ -solvable ASM program.

To any algorithm A that is ψ -definable there corresponds an equivalent ψ -abstract state machine B , and hence a ψ -solvable program Π . Conversely, a ψ -abstract state machine B corresponds to a ψ -definable algorithm. However, not every program Π is ψ -solvable.

When ψ corresponds to ψ_{der} , unambiguity comes from (unicity in) the Cauchy-Lipschitz theorem. The fact that not every program Π is ψ -solvable is due to the fact that not all differential equations have a solution.

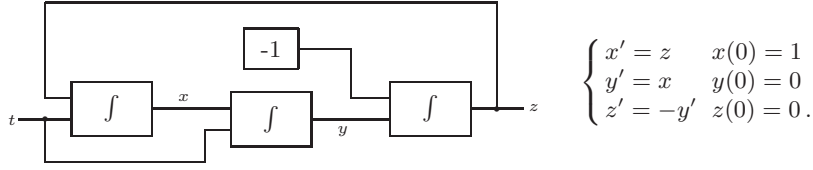


Fig. 1. A GPAC for sine and cosine (left). Corresponding evolution (right).

6 Examples

The examples in this section are for semantics ψ_{der} . Our settings cover, first of all, analog algorithms that are pure flow, in particular all systems that can be modeled as ordinary differential equations. A very simple, classical example is the pendulum: the motion of an idealized simple pendulum is governed by the second-order differential equation $\theta'' + \frac{g}{L}\theta = 0$, where θ is angular displacement, g is gravitational acceleration, and L is the length of the pendulum rod. This can indeed be modeled as the program

```
flow
  DYNAMIC( $\theta, \theta_1$ )
  DYNAMIC( $\theta_1, -\frac{g}{L} \cdot \theta$ )
endflow
```

using the fact that any ordinary differential equation can be put in the form of a vectorial first-order equation, here θ_1 corresponding to the derivative of θ .

As a consequence, our formalism covers very generic classes of continuous-time models of computation, including the GPAC, which corresponds to ordinary differential equations with polynomial right-hand sides [13,12]. Recall that the GPAC was proposed as a mathematical model of differential analyzers (DAs), one of the most famous analog computer machines in history. Figure 1 (left) depicts a (non-minimal) GPAC that generates sine and cosine. In this picture, \int signifies some integrator, and -1 denotes some constant block. This simple GPAC can be modeled by the program

```
flow
  DYNAMIC( $x, z$ )
  DYNAMIC( $y, x$ )
  DYNAMIC( $z, -x$ )
endflow
```

Our proposed model can also adequately describe hybrid systems, made of alternating sequences of continuous evolution and discrete transitions. This includes, for example, a simple model of a bouncing ball, the physics of which are given by the flow equations $x'' = -gm$, where g is the gravitational constant and $v = x'$ is the velocity, except that upon impact, each time $x = 0$, the velocity changes according to $v' = -k \cdot v'$, where k is the coefficient of impact. Every time the ball bounces, its speed is reduced by a factor k . This system can be described by a program like

```

if  $x = 0$  then
   $v := -k \cdot v$ 
else
  flow
    DYNAMIC( $x, v$ )
    DYNAMIC( $v, -g.m$ )
  endflow
endif

```

Our setting is an extension of classical discrete-time algorithms; hence, all classical discrete-time algorithms can also be modeled.

As for examples with semantics other than ψ_{der} : Observe that one can consider timelines like \mathbb{Q} instead of \mathbb{R} . (For such a timeline, we have $\text{Flow}(i)$ for all $i \in \mathbb{Q}$.) One can define a semantics on such a timeline where for every state X we have $\text{Flow}(X)$ by first extending the evolution function to \mathbb{R} (for example by restricting to continuous dynamics) and then using the derivative. Constructions of [19] are also covered by our settings: In some sense, the example at the beginning of the paragraph is the spirit of the constructions from [19], where the timeline is the set of hyperreals obtained by multiplying some fixed infinitesimal by some hyperinteger (using hyperreals and infinitesimals). Notice that there is no need to consider derivatives or similar notions: we could also consider analytic dynamics, and consider a semantics related to the family of Taylor coefficients. Weaker notions of solution, like variational approaches, can also be considered.

References

1. Blum, L., Shub, M., Smale, S.: On a theory of computation and complexity over the real numbers; NP completeness, recursive functions and universal machines. *Bulletin of the American Mathematical Society* **21** (1989) 1–46
2. Boker, U., Dershowitz, N.: The Church-Turing thesis over arbitrary domains. In: *Pillars of Computer Science. Lecture Notes in Computer Science*, Vol. 4800. Springer (2008) 199–229
3. Boker, U., Dershowitz, N.: Three paths to effectiveness. In: *Fields of Logic and Computation*. Springer (2010) 135–146
4. Bournez, O., Campagnolo, M.L.: A survey on continuous time computations. In: *New Computational Paradigms. Changing Conceptions of What is Computable*. Springer (2008) 383–423
5. Bournez, O., Dershowitz, N., Falkovich, E.: Towards an axiomatization of simple analog algorithms. In: *Proc. 9th Annual Conference on Theory and Applications of Models of Computation*. Springer (2012) 525–536
6. Bournez, O., Dershowitz, N., Néron, P.: Axiomatizing Analog Algorithms. *ArXiv e-prints* <http://arxiv.org/abs/1604.04295> (2016)
7. Bush, V.: The differential analyser. *Journal of the Franklin Institute* **212** (1931) 447–488
8. Cohen, J., Slissenko, A.: On implementations of instantaneous actions real-time ASM by ASM with delays. In: *Proc. 12th Intl. Workshop on Abstract State Machines*. Université de Paris 12 (2005) 387–396

9. Cohen, J., Slissenko, A.: Implementation of sturdy real-time abstract state machines by machines with delays. In: Proc. 6th Intl. Conf. on Computer Science and Information Technology. National Academy of Science of Armenia (2007)
10. Dershowitz, N., Gurevich, Y.: A natural axiomatization of computability and proof of Church's Thesis. *The Bulletin of Symbolic Logic* **14** (2008) 299–350
11. Fu, M.Q., Zucker, J.: Models of computation for partial functions on the reals. *J. Logical and Algebraic Methods in Programming* **84** (2015) 218–237
12. Graça, D.S., Buescu, J., Campagnolo, M.L.: Computational bounds on polynomial differential equations. *Appl. Math. Comput.* **215** (2009) 1375–1385
13. Graça, D.S., Costa, J.F.: Analog computers and recursive functions over the reals. *Journal of Complexity* **19** (2003) 644–664
14. Gurevich, Y.: Sequential abstract-state machines capture sequential algorithms. *ACM Trans. Comput. Log.* **1** (2000) 77–111
15. Hasuo, I., Suenaga, K.: Exercises in nonstandard static analysis of hybrid systems. In: *Computer Aided Verification*. Springer (2012) 462–478
16. Nyce, J.M.: Guest editor's introduction. *IEEE Ann. Hist. Comput.* **18** (1996) 3–4
17. Platzer, A.: Differential dynamic logic for hybrid systems. *J. Automated Reasoning* **41** (2008) 143–189
18. Reisig, W.: On Gurevich's theorem on sequential algorithms. *Acta Informatica* **39** (2003) 273–305
19. Rust, H.: Hybrid abstract state machines: Using the hyperreals for describing continuous changes in a discrete notation. In: *Intl. Workshop on Abstract State Machines*. Swiss Federal Institute of Technology (2000) 341–356
20. Shannon, C.E.: Mathematical theory of the differential analyser. *Journal of Mathematics and Physics* **20** (1941) 337–354
21. Suenaga, K., Hasuo, I.: Programming with infinitesimals: A while-language for hybrid system modeling. In: *Automata, Languages and Programming*. Springer (2011) 392–403
22. Tucker, J.V., Zucker, J.I.: A network model of analogue computation over metric algebras. In: *New Computational Paradigms*. Springer (2005) 515–529